递归与分治法

全排列问题

- 例3. 排列问题：设计一个递归算法生成n个元素$\{r_1, r_2, \ldots, r_n\}$的全排列。

有些问题表面上不是递归定义的，但可通过分析，抽象出递归的定义

设 $R = \{r_1, r_2, \ldots, r_n\}$是要进行排列的$n$个元素， $R_i = R - \{r_i\}$

集合X中元素的全排列记为$perm(X)$

$perm(X)(r_i)$表示在全排列$perm(X)$的每一个排列后加上后缀r_i得到的排列

R的全排列可归纳定义如下：

当$n = 1$时， $perm(R) = (r)$，其中r是集合R中唯一的元素

当$n > 1$时， $perm(R)$由$perm(R_n)(r_n), perm(R_{n-1})(r_{n-1}), \ldots, perm(R_1)(r_1)$构成

全排列问题(续)

- **要求**：写一个**就地**生成n个元素$a_1, a_2, \ldots, a_n$全排列 $(n!)$ 的算法，要求算法终止时保持$a_1, a_2, \ldots, a_n$原状

  解：设 A[0...n-1] 类型为char，“就地”不允许使用 A 以外的数组

  ① 生成$a_1, a_2, \ldots, a_n$全排列**分割为n个子问题**：求n -1个元素的全排列 + n^{th}个元素

1st子问题	$a_1, a_2, \ldots, a_{n-1}$	a_n	//
2nd子问题	$a_1, \ldots, a_{n-2}, a_n$	a_{n-1}	//swap A[n-2], A[n-1]
3rd子问题	$a_1, \ldots, a_n, a_{n-1}$	a_{n-2}	//swap A[n-3] ,A[n-1]
⋮	⋮	⋮	
nth子问题	$a_n, a_2, \ldots, a_{n-1}$	a_1	//swap A[0],A[n-1]

  ② 递归终结分支

  当 n=1 时， 一个元素全排列只有一种，即为本身。实际上无须进一步递归，可直接打印输出A

全排列问题(续)

③ **算法：**

```
void  permute(char  A[ ], int  n) {
      if (n==1)  print (A);
      else {
           permute(A,n-1); //求A[1..n-1]的全部排列。1st子问题不用交换
           for (i=n-1; i>=1; i--) {
                Swap(A[i], A[n]);   // 交换a_i和a_n内容
                permute(A,n-1);     // 求A[1..n-1] 全排列
                Swap(A[i],A[n]);   //交换，恢复原状
           }//endfor
       }//endif
}
```

整数划分

- 例4. 整数划分：将正整数n表示成一系列正整数之和：$n = n_1 + n_2 + \cdots + n_k$，其中$n_1 \geq n_2 \geq \cdots \geq nk \geq 1$，$k \geq 1$。正整数$n$的这种表示称为正整数$n$的划分。求正整数$n$的不同划分个数。

例如，正整数6有如下11种不同的划分：

6；

5+1；

4+2，4+1+1；

3+3，3+2+1，3+1+1+1；

2+2+2，2+2+1+1，2+1+1+1+1；

1+1+1+1+1+1。

整数划分(续)

- 在本例中，设$P(n)$为正整数n的划分数，难以直接找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作$q(n,m)$。

  可以建立$q(n,m)$的如下递归关系：

  - $q(n,1)$ =1，n≥1;

    当最大加数n_1不大于1时，任何正整数n只有一种划分形式，即n =1+1+……+1

  - $q(n,m) = q(n,n)$，m≥n;

    最大加数n_1实际上不能大于n, $q(1,m) = 1$

  - $q(n,n)$ =1+ $q(n,n-1)$;

    正整数n的划分由$n_1 = n$的划分和$n_1 \leq n$ -1的划分组成。

  - $q(n,m) = q(n,m-1) + q(n-m,m)$，$n > m > 1$;

    正整数n的最大加数n_1不大于m的划分由$n_1 = m$的划分和$n_1 \leq m-1$的划分组成。

整数划分(续)

- **递归函数：**

  正整数n的划分数：$p(n)=q(n,n)$

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & n<m\\ 1+q(n,n-1) & n=m\\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

- **代码实现：**

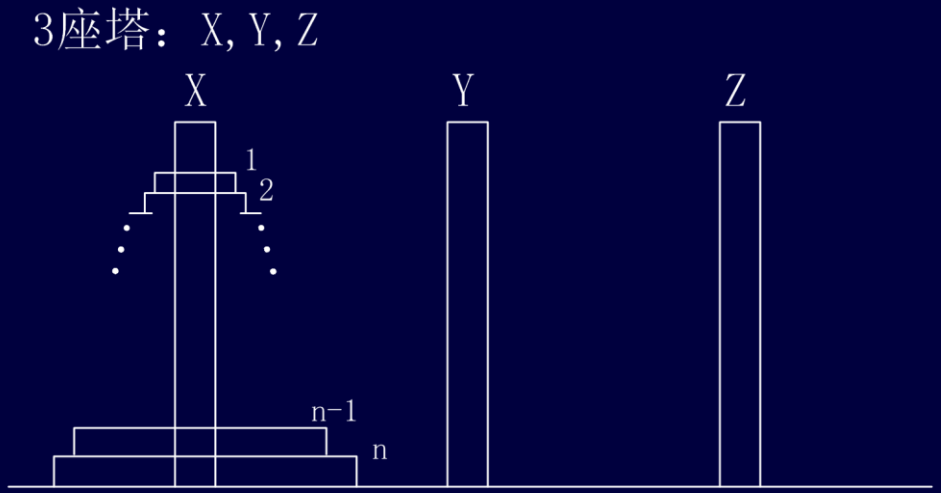
```
int q(int n, int m){
    if ( (n<1) || (m<1) ) return 0;
    if ( (n==1) || (m==1) ) return 1;
    if (n<m) return q(n,n);
    if (n==m) return q(n,m-1)+1;
    return q(n,m-1)+q(n-m,m);
}
```

汉诺塔问题

- 例5. 汉诺塔问题：

  将X上的圆盘移到Z上，要求按同样次序排列，且满足：

  1. 每次只能移动一片
  2. 圆盘可插在X,Y,Z任一塔座上
  3. 任一时刻大盘不能压在小盘上



汉诺塔问题

- **问题分析：**

n阶Hanoi塔问题$Hanoi(n, x, y, z)$

当$n = 0$时，没盘子可供移动，什么也不做

当$n = 1$时，可直接将1号盘子从x轴移动到z轴上

当$n = 2$时，先将1号盘子移动到y轴，再将2号盘子移动到z轴，最后将1号盘子移动到z轴

对于一般$n > 0$的一般情况可采用如下分治策略进行移动：

(1)将1至n-1号盘从x轴移动至y轴，可递归求解$Hanoi(n\text{-}1, x, z, y)$

(2)将n号盘从x轴移动至z轴

(3)将1至n-1号盘从y轴移动至z轴，可递归求解$Hanoi(n\text{-}1, y, x, z)$

汉诺塔问题(续)

思考：四个塔的情况需要怎么移动盘子呢？

原问题：将n片盘子从X移到Z，Y为辅助塔，可分解为：

1. 将上面n-1 个盘从X移至Y，Z为辅助盘
2. 将 n^{th} 片从X移至 Z
3. 将Y上n-1个盘子移至Z，X为辅助盘
4. 终止条件： n = 1时，直接将编号为1的盘子从 X 移到Z

递归算法：

```
void Hanoi(int n, char x, char y, char z ) {// n个盘子从X移至Z， Y为辅助
    if ( n==1 )  move(x,1,z); // 将1号盘子从X移至Z, 打印
    else {
        Hanoi (n-1,x,z,y);  //源X， 辅Z， 目Y
        move (x,n,z);       // 将n号盘子从X移至Z
        Hanoi (n-1,y,x,z);  //源Y， 辅X， 目Z
        }
    }
```

汉诺塔问题(续)

- **性能分析：**

设$T(n)$表示n个圆盘的汉诺塔问题移动圆盘的次数，那么显然$T(n)$ =0，对于n>0的一般情况采用如下分治策略：

(1)将1至n-1号盘从X轴移动至Y轴，可递归求解$Hanoi(n\text{-}1, X, Z, Y)$；

(2)将n号盘从X轴移动至Z轴；

(3)将1至n-1号盘从Y轴移动至Z轴，可递归求解$Hanoi(n\text{-}1, Y, X, Z)$。

在(1)与(3)中需要移动圆盘次数$T(n-1)$，(2)需要移动一次圆盘。可得如下关系：

$$T(n) = 2T(n-1) + 1$$

展开上式可得：

$$\begin{aligned} T(n) &= 2 * T(n-1) + 1 \\ &= 2 * [2 * T(n-1) + 1] + 1 = 2 * 2 * T(n-1)) + 1 + 2 \\ &\cdots\cdots \\ &= 2^n T(n-1) + 1 + 2 + \cdots + 2^n \\ &= 2^n + 2^{n+1} - 1 \end{aligned}$$

时间复杂度：$O(2^n)$

分治算法时间性能分析

- **性能分析：**
  - 设 $T(n)$是规模为n的一个问题的执行时间，若问题规模足够小，如n<c(c为常数)，则直接求解的时间为$\Theta(1)$
  - 设分解问题成子问题需要的时间为$D(n)$
  - 设原问题被分解为a个子问题，每个子问题的规模为原问题的$1/b$，则求解各子问题的时间为$aT(n/b)$
  - 设合并子问题的解成原问题的解需要时间为$C(n)$，则得到递归式：

$$T(n)=\begin{cases}\Theta(1) & \text{若 } n\le c\\ aT(n/b)+D(n)+C(n) & \text{其他}\end{cases}$$

注意：在用分治法设计算法时，最好使子问题的规模大致相同，这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。

递归和分治总结

- **优点**：结构清晰，代码可读性强，为设计算法、调试程序带来很大方便
- **缺点**：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。
- **解决方法**：
  - 使用尾递归/分治法
  - 用递推来实现递归函数(转化为非递归)
  - 采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显